Finding Dependency, Test Sequences and Test Cases for Simulink/Stateflow Models

Ravikant Sharma

Roll. 213CS3177

under the supervision of

Prof. Durga Prasad Mohapatra

Department of Computer Science and Engineering

National Institute of Technology Rourkela

Rourkela – 769008, India

Finding Dependency, Test Sequences and Test Cases for Simulink/Stateflow Models

Dissertation submitted in

May 31st 2015

to the department of

Computer Science and Engineering

of

National Institute of Technology Rourkela

in partial fulfillment of the requirements

for the degree of

Master of Technology

by

Ravikant Sharma

(Roll. 213CS3177)

under the supervision of

Prof. Durga Prasad Mohapatra

Department of Computer Science and Engineering

National Institute of Technology Rourkela

Rourkela – 769 008, India

Computer Science and Engineering

National Institute of Technology Rourkela

Rourkela-769 008, India. www.nitrkl.ac.in

June 1, 2015

Certificate

This is to certify that the work in the thesis entitled *Finding Dependency, Test Sequences and Test Cases for Simulink/Stateflow Models* by *Ravikant Sharma*, having roll number 213CS3177, is a record of an original research work carried out by him under my supervision and guidance in partial fulfillment of the requirements for the award of the degree of *Master of Technology* in *Computer Science and Engineering Department*. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Dr. Durga Prasad Mohapatra

Associate Professor

Department of CSE

NIT, Rourkela

Acknowledgment

First of all, I would like to express my profound feeling of admiration, my token of gratitude and appreciation towards my supervisor Prof. Durga Prasad Mohapatra, who has been the controlling and guiding force behind this work. I need to express gratitude towards him for acquainting me to the field of Software testing and Slicing and giving me the chance to work under him. His unified confidence in this point and capacity to draw out the best of explanatory and viable aptitudes in individuals has been significant in extreme periods. Without his important guidance and help it would not have been workable for me to finish this thesis work. I am significantly obligated to him for his consistent support and precious guidance in every part of my scholarly life. I think of it as my favorable luck to have got a chance to work with such a wonderful person.

I thank our H.O.D. Prof. Santanu Kumar Rath for their steady backing in my thesis work. He has been incredible wellsprings of motivation to me and I say thanks to him in the name of all that is pure.

I would also like to thanks my all lab mates specially Rohan Koshy for helping and providing me suggestions. I would also like to thanks my all friends, classmate and PHD scholars for their supports, encouragements and helps whether direct or indirect during my thesis work.

Last but not the least I dedicated my thesis work to my parents and siblings for their constant supports and motivations regularly during my hard times.

I wish to thank all the faculty members and secretarial staff of the CSE Department for their thoughtful collaboration and helps.

Ravikant Sharma

Abstract

The Simulink/Stateflow (SL/SF) acquiring from Mathworks is fitting the de facto standard in industry for model based development especially for embedded control systems. Many industrial tools are available in the market for test case generation from SL/SF designs; though, we have observed that these tools do not accomplish satisfactory coverage in cases when designs involve non-linear blocks and Stateflow blocks transpire deeper inside the Simulink blocks. For this purpose, we have proposed a methodology that generates the test sequences and test cases from the Simulink/Stateflow model. In our approach, first, we have developed a SL/SF model using MATLAB tool which generates mdl(model description language) file. Next, we convert that mdl file into xml file and then the xml file and mdl file path are passed as an inputs to our proposed methodology to generate Simulink/Stateflow dependency graph(SSDG); Now using the SSDG, we generate test sequences by applying depth first search approach(DFS). Next, for each test sequence, we generate a set of test cases and finally we prioritize those test cases using information flow(IF)value.

Now a days, Simulink/Stateflow models become the de-facto standard in the modelling of control system based development of real-time system, especially for an embedded system. These are extensively used in many domains, including automotive and avionics. It allows modelling the systems, simulating and analyzing dynamic systems. The resultant Simulink/Stateflow models consist of large numbers of blocks and states likes more than ten thousand blocks. Hence, to certify the quality of such control system models, automated static analyses and slicing methods are necessary to deal with this complexity. Hence, these approaches help in debugging the model, understanding the behaviour of models,identifying faults,if occurs. In this thesis, we present an approach for computing intradependencies between blocks by the concept of slicing approach and we represent the result using dependence graphs. With the help of slicing approach, the complexity of a system model can be compact to a specified point of interest(Slicing Criterion) concern by removing unrelated blocks in model system.

Keywords: SL/SF Model, Dependency Graph, Test Sequences, Test Cases, SSDG, Model based testing, Slices, Forward Slicing, Backward slicing

Contents

List of Figures

Chapter 1

Introduction

In this section we discuss introduction of our work, motivations about our works, objective of our works and thesis organisation of our works.

1.1 Introduction

Every software product goes through a different life state during the development i.e. System undergoes the changes during life cycles,these changes occur due to various reasons such as improvement and modification in the existing functionalities,detecting fault and defects in the model system. Every time whenever the changes occur in the software model, the changed software model is to be tested and verified so that the modified code as well as design does not negatively affect the behavior of unmodified code and design.Due to changes,the software becomes complex during testing, so the use of proper design models for software tasks has become important. Models of a system represents the needed behavior of the system or to represent an approach for testing and we can test this model through model based testing. Therefore, we need a formal verification of the models for detecting defects and faults.Quality assurance is an important issue for software development.

The Simulink/Stateflow (SL/SF) upbringing from Mathworks is fitting the accepted standard in industry for model based advancement, particularly for

installed control frameworks. Numerous mechanical apparatuses are accessible in the business sector for experiment era from SL/SF plans; however, we have watched that these instruments don't fulfill coverage scope in situations when outlines include non-direct blocks and Stateflow blocks come to pass more profound inside the Simulink model. For this reason, we have proposed a strategy that creates the test sequence generation and test cases generation from the Simulink/Stateflow model.

1.2 Motivation

Matlab Simulink/Stateflow(SL/SF) model now becomes an inherent parts of many applications for embedded system. Matlab SL/SF is used especially for developing embedded systems for complex and composite systems.

SL/SF Model helps in modelling systems, even if they are more complex contains large number of blocks.The resulting model must be tested and validated in order to detect faults and defects in the system.But, such model consists of a large number of blocks in the systems, due to which testing process becomes difficult to test.

Presently a days Simulink/Stateflow models turns into the true standard in the displaying of control framework based advancement of ongoing, particularly for an installed framework. These are widely utilized as a part of numerous areas, including automotive and avionics. It empowers displaying the frameworks, mimicking and examining element frameworks. The resultant Simulink/Stateflow models comprise of huge quantities of blocks and states consists of more than ten thousand blocks. Consequently, to guarantee the nature of such control frameworks models, computerized static examinations and slicing approach systems are important to manage up with these complexity.

1.3 Objectives of our thesis:

- To propose an algorithm that can generate dependency graph for Simulink/Stateflow model.
- To propose an algorithm that can generate test sequences for Simulink/Stateflow models using the proposed dependency graph.
- To generate the test cases for each and every test sequences for Simulink/Stateflow models.
- To propose an algorithm that prioritize the test cases.
- To propose an algorithm that generate the forward slicing dependency graph.
- To propose an algorithm that generate the backward slicing dependency graph.

1.4 Organization of the thesis

The rest of the thesis is organized as follows:

- Chapter 2: In this section we present the basic concepts and concept requires in our work. We discuss about testing, testing technique, simulink/stateflow models, Model based testing, dependency graph, etc
- Chapter 3: In this chapter we present litrature review related to simulink/stateflow models testing and slicing.
- Chapter 4: In this chapter we first discuss our proposed methodology of test case and test sequence generation for simulink/stateflow model, next we show implematation along with one case study and result.
- Chapter 5: In this section, we first discuss about proposed model of computing dependency betweens blocks of Sl/SF model using slicing approach, algorithm

used and next we discuss about their implementation along with exmaple and then shows result as dependency graph.

- Chapter 6: At last we concluded our work and discuss few future works possible on this area.

Chapter 2

Basic Concepts and Definitions

2.1 Fundamentals of Software Testing

Software Testing

Software testing is the process of finding bugs or defects in the software by executing a program. Software Testing can also be viewed as the process of verifying and validating the software products or application or program that meets the requirements of the software and design of the software, works as the expectation of the software.

- **White-Box Testing**: In this type of testing everything has been shown that is way it is called as glass box Testing or structural testing. Here tester is worried about to find the paths from the code and induce the output.It is based on source code before integration.It applies in every level of testing expect user level. The basic objective is to map which line of code will produce which correct output.

- **Black-Box Testing:** This testing is a technique for programming testing that inspects the usefulness of an application without peering into its inside structures or workings. This technique for test can be connected to basically every level of programming testing: unit, reconciliation, framework

and acknowledgment.Particular information of the application's code/inward structure and programming learning when all is said in done is not needed.

Software testing can be dividing mainly into two parts:

Static testing: Inspection, walkthrough, reviewing, etc are the example static testing that can be evaluated for finding defects without executing any coding. It a manual testing. Static testing is done during the process of verification. Static testing includes the static analysis and reviewing of design and source code for finding error or defects in the software.

Dynamic testing: Dynamic testing is done during the process of validation. Dynamic testing is when you are working with the actual system (not some artifact or model that represents the system), providing an input, receiving an output and comparing the output to the expected behavior. It is hands-on working with the system with the intent of finding errors.

2.2 Different levels of testing

In this present reality it is difficult to give 100% effective programming testing. But by the assistance of a powerful testing it is conceivable to bear the cost of high level of enlistment of testing. The path in which we are distinguishing the experiments is known as programming, testing techniques. The essential goal of the experiments is to discover a great number of bugs and to cover the expansive area. Testing is done all over of the software development life cycle phase, however, it carries on a contrastingly in distinctive circumstances. There are diverse levels of testing exist, for example, Unit validation Testing, Integration Testing, System Testing, Acceptance Testing, Regression Testing.

2.2.1 Unit Testing

This kind of testing is performed by specialists before the setup is offered over to the testing gathering to formally execute the software. Unit testing is performed by the

specific testers on the individual units/mosule of source code doled out reaches. The testers usage test data that is special in connection to the test data of the quality affirmation bunch.

The target of unit testing is to detach every piece of the module and exhibit that individual parts are cure similarly as requirements and handiness.

Limitation of Unit Testing

Testing can't get every last bug in an application. It is difficult to assess each execution way in every product application. The same is the situation with unit testing.

There is a cutoff to the quantity of situations and test information that a designer can use to check a source code. In the wake of having depleted all the choices, there is no decision yet to stop unit testing and union the code portion with different units.

2.2.2 Integration Testing

Integration testing is portrayed as the testing of joined parts of an application to make sense of whether they work viably. Coordination testing could be conceivable in two courses: Bottom up integration testing and Top-down integration testing.

1. Bottom-up integration testing

This testing begins with unit testing, trailed by tests of alterably more lifted sum mixes of units called modules or develops.

2. Top-down integration

In this testing, the most hoisted sum modules are attempted first and sensibly, lower-level modules are attempted from that point on.

2.2.3 System testing

System testing tests the framework all in all. At the point when all the parts are composed, the application general is attempted altogether to see that it meets the foreordained Quality Standards. This kind of testing is performed by a specific tester group.

System testing is critical due to the accompanying reasons:

- System testing is the initial phase in the Software Development Life Cycle, where the application is tried all in all.
- The application is attempted by and large to affirm that it meets the utilitarian and specific subtle elements.
- The application is tried in a situation that is near to the generation environment where the application will be conveyed.
- System testing empowers us to test, confirm, and accept both the business prerequisites and in addition the application structural planning.

2.2.4 Regression Testing

At whatever point an adjustment in a product application is made, it is very conceivable that different zones inside the application have been influenced by this change. Regression testing is performed to confirm that an altered bug hasn't brought about another usefulness or business principle infringement. The goal of regression testing is to guarantee that a change, for example, a bug fix ought not bring about another deficiency being uncovered in the application.

Regression testing is critical on account of the accompanying reasons:

- Minimize the holes in testing when an application with changes made must be tried.

- Testing the new changes to confirm that the progressions made did not influence whatever other territory of the application.
- Mitigates dangers when regression testing is performed on the application.
- Test scope is expanded without bargaining courses of events.
- Increment rate to market the item.

2.2.5 Acceptance Testing

This is apparently the most vital sort of testing, as it is led by the Quality Assurance Team who will gauge whether the application meets the expected details and fulfills the customer's prerequisite. The QA group will have an arrangement of pre-written situations and experiments that will be utilized to test the application.

More thoughts will be imparted about the application and more tests can be performed on it to gauge its exactness and the reasons why the venture was started. Acceptance tests are not just expected to bring up basic spelling oversights, corrective mistakes, or interface holes, additionally to call attention to any bugs in the application that will bring about framework crashes or significant slips in the application.

By performing acceptance tests on an application, the testing group will reason how the application will perform underway. There are likewise legitimate and contractual prerequisites for acceptance of the framework.

2.3 Test Case and Test Sequenece

Test case is a triplet which contains input, state of the system and output. Whereas test sequence is the flow of execution. Path testing what it gives at the end flow of execution. In the same way test sequence is also flow of execution in which order execution flows.

2.4 Model Based Testing

Model-based testing is a use of model-based outline for designing and alternatively additionally executing relics to perform software testing or system testing. Models can be utilized to constitute desired behavior of a System Under Test (SUT), or to constitute to test systems and a testing situation. Model-based testing is often a systematic method to get test cases from types of system requirements. It permits you to assess necessities autonomous of algorithmic design and development.

2.5 Simulink/Stateflow Introduction

Simulink/Stateflow is developed by "The Math works". The Developer "The Math works" describes it as "as a platform for multidomain simulation of model based design for dynamic system".

Matlab Simulink/Stateflow(SL/SF) model now becomes an inherent parts of many applications for embedded system. Matlab SL/SF is used especially for developing embedded systems for complex and composite systems.

SL/SF Model helps in modelling systems, even if they are more complex contains large number of blocks.The resulting model must be tested and validated in order to detect faults and defects in the system.But, such model consists of a large number of blocks in the systems, due to which testing process becomes difficult to test.

2.5.1 Simulink

The Simulink library provides a dynamic graphical interface with a custom set of block libraries that are useful in design, simulation, implementation, test coverage, verification and validation of model based testing especially embedded system.

Basically Simulink model can be composed of different sets of predefined blocks in the simulink library. These simulink Blocks are organized according to the behavior into a customized blocks inside the simulink library. The important library blocks

contain the following as:

- **Source library:** It contains blocks that are used for generating the signals. Ex : Constant, Sine wave, ground, inport, clock, ramp, signal builder, signal generator, etc.
- **Sink library:** It contains blocks that are useful for display result or output write block. Ex: outport, scope, terminator, display, floating scope, stop simulation, etc.
- **Continuous library:** It contains the block that defines continuous state. Ex: Derivative, integrator, transport delay, variable time delayed.
- **Discrete Library:** It contains blocks that define discrete states and discrete time components. Ex: Discrete-time integrator, unit delay, difference, delay, discrete filter, discrete PID controller, etc.
- **Math operation library:** Its contains blocks that are useful for representing mathematical operations. Ex: gain, sum, product, Dot product, etc.
- **Discontinuous library:** It contains blocks that define dis-continuous states. Ex: Saturation, Quantizer, rate limiter, wrap to zero, etc
- **Logic and bit operation library:** It contains library that represents and perform the logical and bit operations. Ex: logical operator, bitwise operator, relational operators, bit set, bit clear,etc
- **Lookup table library:** It contains blocks that model the non-linearity with lookup tables. Ex: 1-D lookup table, 2-D Lookup Table, cosine, sine, etc
- **Signal attributes library:** It contains blocks that support signal attributes. Ex Data Type Conversion, Rate Transition, Signal Conversion, IC
- **User Defined function:** Its contains blocks that supports user defined custom functions. Ex Argument Inport, Argument Outport, Fcn, Function Caller, etc

2.5.2 Stateflow

Stateflow represents the state behavior of the system. It provides the language elements that are integrated with simulink required to represents complex state depend behavior. Both simulink and Stateflow work together. Running any one either Stateflow or simulink, automatically runs both simultaneously. It provides the potentiality for designing complex system especially embedded system that contains controls, states, supervisory, and mode logic. Every state chart diagram is represented by state blocks and transition in Stateflow library.

A Stateflow model helps to describe the system behavior using the several use of graphical and non graphical construct. Graphical construct includes states block, transition, and junction and function elements. Non-Graphical construct includes events identifier, condition and condition actions, function calls, etc.

States represents the basic object that reflects the modes of the system behavior. Transition is used to connect them and shows the flow control of the system states. The state can be either active or inactive. Active state means that the Stateflow is in that mode.

Events and conditions cause the state to be change from one state to other i.e. From inactive to active states. There are different types of action that the states of the Simulink/Stateflow support. These are:

- **Entry Action:**It defines what action to be take place when states become active or entered. For example in the figure 1, state first has the entry section x=10, that means when first state become active or entered it automatically sets the value of x to 10.

- **During Actions:** It defines what action is to be take place when state is already active i.e. these action are to executed whenever a particular state is already active and some event other than the specified condition stuff or exit condition of transition occurs. For example in figure 1, state first have the during section of x=x+1 i.e. this action is executed whenever state first is

active and some event occurs then it increments the value of x by 1.

- **Exit section:** It defines what action is to be taken place when states become active to inactive. For example in gure 1, state Second have the exit action of x=10 i.e when states becomes inactive from active state it sets the value of x to 10.
- **On Event Actions:** It defines what action to be take place when state is active and particular mentioned event to be occurred.

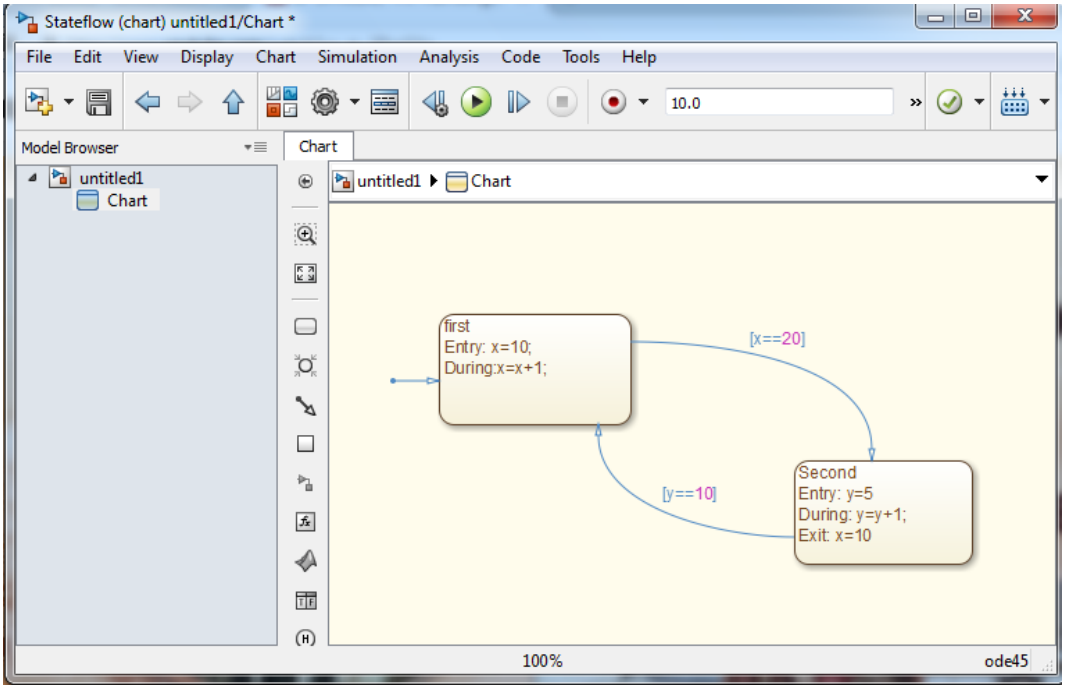

Figure 2.1: Sample Simulink Diagram

2.6 MDL

Simulink/Stateflow model in Matlab tool is saved or stored in the form of .mdl file. The mdl file stands for model description language file. The mdl file is stores the description in the form of structured ASCII format. Stateflow block also store information into the mdl file.

2.7 XML

XML stands for Extensible Markup Language. XML is used to describe data. The XML is the negotiable way to store information formats of electronic data. XML is a text based format that allows for the structuring of electronic documents and is not limited to a set of labels. XML is the markup language that is used for encoding the document in a format that are reliable for human readable. It is easy to understand.It is a format with the powerful support of the Unicode Standard. The Unicode Standard includes an encoding method, set of standard character encodings, set of code charts for viewable reference, etc.

- An intermediate representation of Simulink/stateflow model is an XML file that captures all implicit and explicit dependencies amongs the blocks and states within the model.
- XML format is easy to understand and the model information can be easily understood and retrieved by the use of existing XML parsers algorithm.
- That XML file is further used for converting into the intermediate graph.

2.8 Dependency Graph

The dependency graph having nodes and edges, where nodes represents states of the model and edges represents transition condition or input to the next state.

The dependency graph is used to captures all the implicit dependencies between the blocks of the SL/SF model. So that with the help of dependency graph, we can generates test sequences and test cases for SL/SF model.

Chapter 3

Literature Review

Simulink/Stateflow has originally been designed for the modeling and simulation of dynamic system. Automated test case generation for SL/SF model is necessary for detecting bugs and errors. Many authors have tried different way of approach related to generating test data for Simulink model and verification for Simulink/Stateflow diagram. Many different approaches are there related to our work.

One approach is T-Vec[2] tester that delivers a comprehensive approach for test generation for Simulink model that offers an exhaustive solution for continuous model analysis, test execution and automatic test generation. It drills the path boundaries for generating test vector for path throughout the model hierarchy. Unreachable paths that result in dead code are identified and hyperlinked to the Simulink model elements involved. It is based on the assumption that if there is no coincidental correctness, then test cases that limit the boundaries of domains with arbitrarily high exactitude are adequate to test all the points in the domain. But one disadvantage is that it does not consider the Stateflow of the Simulink.

Clark et. al [3] proposed a technique called tracing and deducing, that enhanced the capability of search-based test data generation for Simulink.

Disadvantage of Zhan and Clark approach: They don't consider a state problem. The state problem remains a challenge for higher level as well as code level test data generation. Automatic generation of test data for higher level models

more generally is a very challenging (performance, reduces as complexity of model increases).

Nayak et. al [4] proposed a methodology Meta model for Simulink/Stateflow called as the Simulink dependency Graph (SDG). The SDG captures all implicit dependencies between different blocks of the SL/SF model and that represents them explicitly, thereby making it possible to perform several types of analysis on the SL / SF model.

MirkoConard et. al [5] proposed an approach that designs a test suite for code generation tools. They describe the design of a test suite for code generation tools. This method provides solutions of different problems of different types of tools that gives how the correct transformation of a source language/model into a target language can be proved. The use of the proposed testing system prompts an era of sets of test suite, which is suitable for testing code generators methodically.

But, the existing code generators can?t guarantee that the automatically generated code from tool, compiles correctly as mentioned in the design due to the following reasons:

1. Errors in the Simulink/Stateflow diagram nodes will get carried over.
2. Errors in the automatic code generator for the Simulink/Stateflow diagram caused for example by finite precision arithmetic or timing constraints.
3. Any human errors in the selection of code generation options, library naming or inclusion, and others.

At the same time, our methodology beats these restrictions, no compelling reason to produce code from the models in our methodology on account of that it beat the MrkoConard's methodology. We also cover all the blocks and all transitions through the generated graph so that our proposed approach over came from these limitations. The Zhan's methodology likewise not covering all the Blocks because of small signal generation, however our methodology defeats this impediment moreover.

Chapter 4

Generation of Test Cases and Test Sequence for SL/SF Models

In this section, firstly we discuss our proposed approach of test cases and a test sequence generation of the Simulink / stateflow model using Simulink/Stateflow dependency graph (SSDG). Next, we discuss proposed algorithm, then their implementation with the help of a case study and finally shows the result.

4.1 Proposed Approach

Our proposed work is based on the graph called Simulink/Stateflow dependency graph (SSDG), in which nodes represent the blocks of the SL/SF model and edges represents the dependencies between blocks. The overall algorithm of our work is as:

Overall Steps of our approach

Step1: Draw Simulink Model by using MATLAB, Simulink tool and Stateflow model is added to the Simulink model by using the MATLAB Stateflow design tool. (It creates.mdl file)

Step 2: Generate XML file for Simulink Model for the above .mdl file.

Step 3: Read the Blocks of model (in Java using.mdl file path as input) and using

an xml parser (xml file as an input), generate an adjacency matrix that contains a dependency amongst the blocks of SL/SF model and store on dotty file.

Step 4: Using dotty file generated in Step3, generate an intermediate graph called Simulink / Stateflow Dependency graph (SSDG) using GraphViz tool.

Step 5: Generate test Sequences using the intermediate graph by applying Depth first search (DFS)approach in the graph.

Step 6: Next, for each test sequence, generate a test case using intermediate graph and dotty file.

Step 7: Prioritize test cases using information flow value(IF).

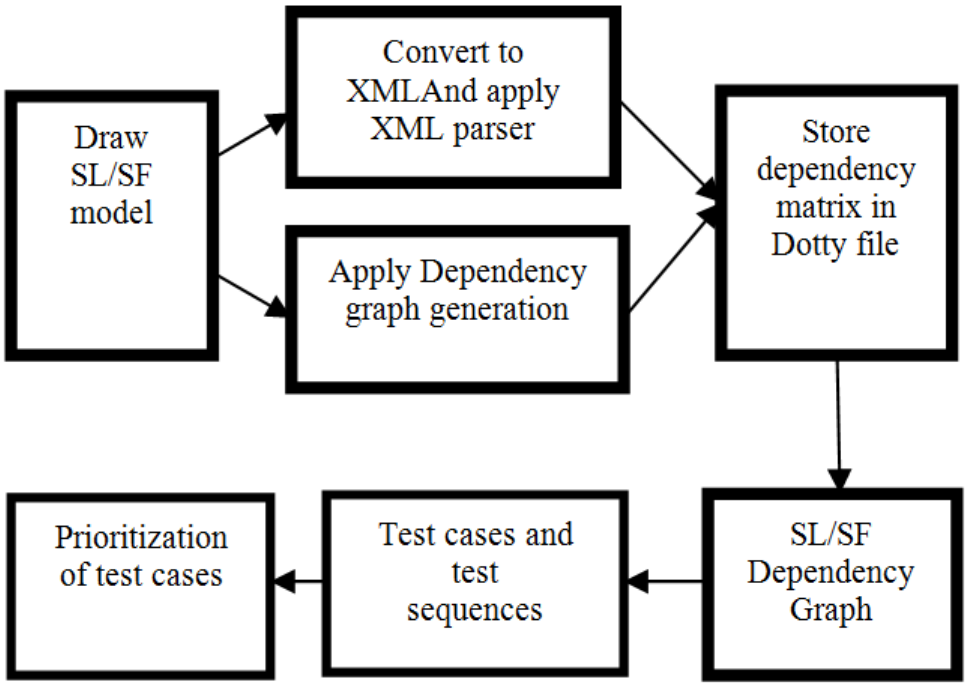

Figure 4.1: Block diagram of our proposed approach for generating test cases for SL/SF model

Figure 4.1 shows our proposed methodology and step by step procedure for generating test cases and test sequences using dependency graph. Firstly, we have to create SL / SF models using Matlab, Simulink library tool, save it to .mdl file. Next

we have to convert that .mdl file into xml file. Now using our proposed algorithm dependency Graph Generation that takes .mdl file and xml file as input and generates SSDG using the GraphViz tool (that takes a dotty file as an input). Next, apply different testing coverage criteria like state coverage and transition coverage to generate test sequences by applying depth first search approaches (DFS). Next, for each test sequences generate set of test cases.Consequently prioritize the generated test cases by prioritization approach.

The algorithm 1 and algorithm 2 are showing the detailed algorithm for dependency graph generation and test sequences genetaion for Simulink/Stateflow model repectively.

4.2 Implementation and Results for a case study

4.2.1 Case study: Automatic Washing Machine

In this section we have taken an example of case study Automatic washing machine for generating test cases, test sequences from a Simulink / Stateflow model for washing machine. So let's discuss our implementation of case study step by step:

Construction of Simulink/Stateflow model:

By Using Mathwork Matlab Tool, Simulink library is present, using this library we have to develop a model by drag and drop the blocks from the design panel of Simulink. StateChart design is also available in the design panel.

Figure4.2 represents the Simulink model for washing machine, in which we are using one signal builder for generating the signal, two constant blocks that represents one for the setting time duration for water fill, washing and drying as the data variable. Another constant for setting hot/cold water wash as set-value variable.

One state chart block that further divided into subsystem that can be viewed as

Algorithm 1 Simulink/Stateflow Dependency Graph Construction

start main
//First, create an SL / SF models using Matlab SL/SF design Tools and save it. (It creates .mdl file)
Take .mdl file path and xml file as an input and generate model object and passed it to the function graphGeneration.
for all up to all block present in SL/SF model **do**
    read each block
    obtain all the neighbor next block of present block
    Write the next block in the matrix form and store that into dot file.
    if any present block contains Simulink/stateflow subsystem **then**
        Push that block into the queue.
    end if
end for
for all up to all block present in queue i.e queue is not empty **do**
    read each block
    obtain all the neighbour next block of present block
    Write the next block in the matrix form and store that into dot file.
end for
Now using GraphViz Tool, take above dot file generated in above step as in input to tool.
Generate Graph using GraphViz tool. This Graph is Called SL/SF depenedency Graph(SSDG).
End main

Algorithm 2 testSequenceGeneration

```
1: Start
2: Take Dependency graph root node v as an input.
3: for all node, Set visited to zero(false).
4: if visited(v)== false then
5:     set visited to true.
6: end if
7: for Each vertex 'w', adjacency of 'v' do
8:     if Not visited(w) then
9:         Call testSequenceGeneration
10:    end if
11: end for
12: End
```

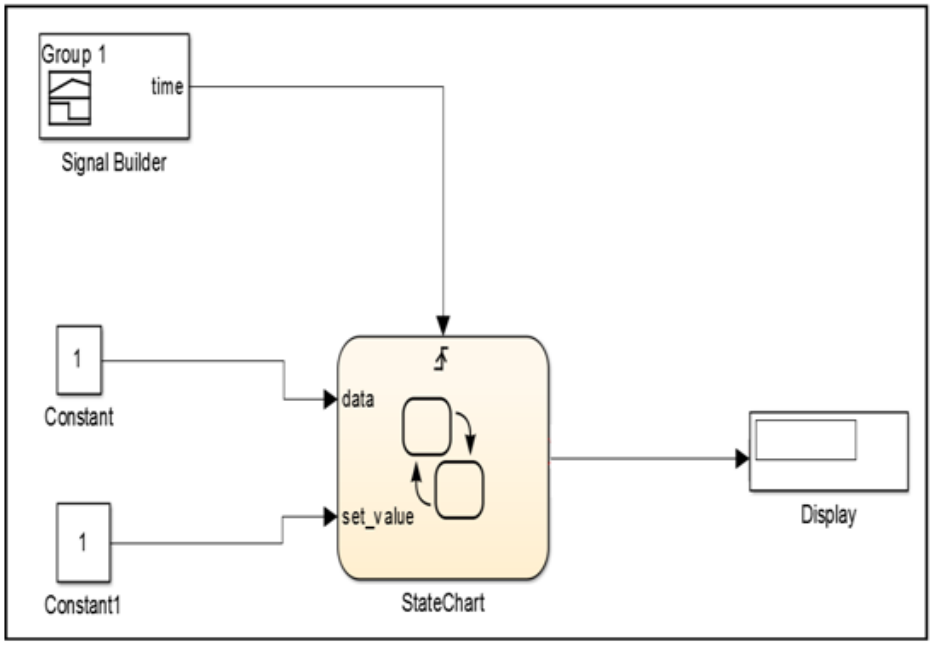

Figure 4.2: Simulink Model for Automatic washing machine

different states of the system. This state chart is present inside the Simulink Platte so this contains both Simulink and Stateflow that why it is called Simulink Stateflow model. Running either one of them will run both simultaneously. One scope block at the end is used for displaying the result of the simulation.

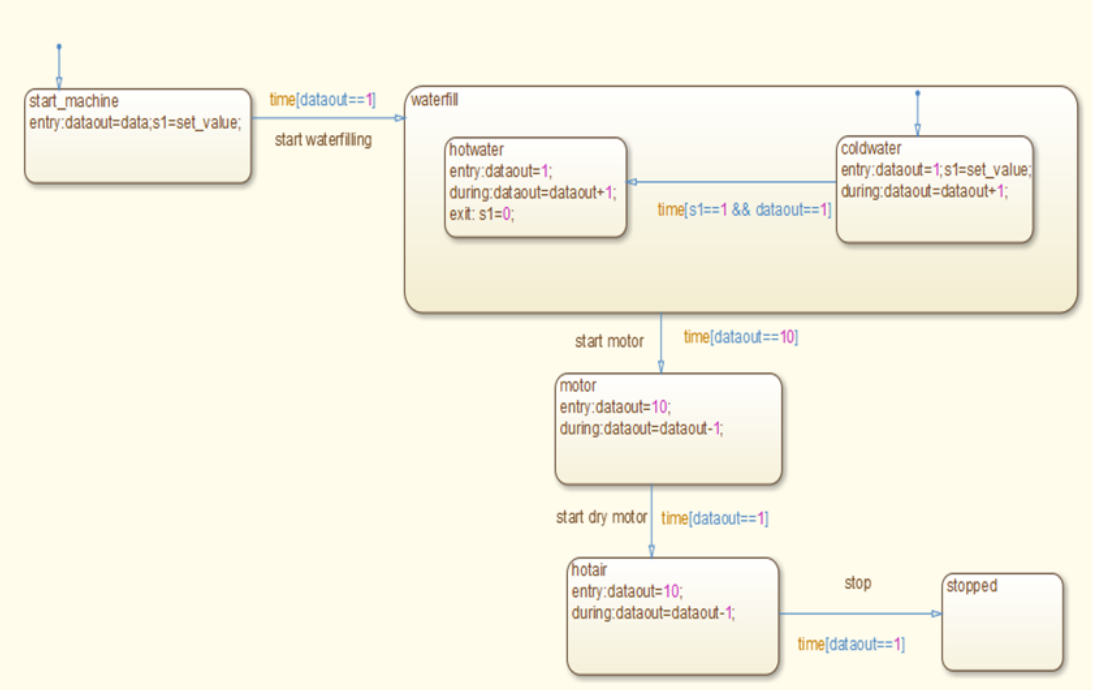

Figure 4.3: State Chart for Automatic washing machine

Figure 4.3 represents the statechart flow chart diagram for the state chart in Figure 4.2. The state chart contains different state present in the system. The different states of the washing machine model are start-machine, water fill, motor, hot air and stopped. This statechart contains two variables, dataout variable used for setting time for different operation in different states and s1 variable for setting hot/cold water wash. Start-machine state contains the entry section in which dataout variable is set as a data constant value and s1 variable is set as set-value where the value 0 represents cold water and the value 1 represents hot water. When start-machine state has found the condition time [dataout==1] is true then only control flow enter into water fill state. Water fill state itself contains two

different states hotwater and coldwater State coldwater contains entry section in which dataout is initialized to 1, during section increments the value by 1 each time treated as the increments in minute during these state. But if condition s1==1 is found to be true then without any operation controls shifted from cold water to hot water state. State hotwater also contains entry section in which dataout is initialized to 1, during section increments the value by 1 treated as the increments in minute during these state.

When either of coldwater and hotwater state found the condition time[dataout==10] to be true, then controls flows transfer to motor state. Initially dataout=10 while entering into motor state that are initialized in entry section, but during this state it decrements the value by 1, upto dataout =1, thats treated as complete 10 min in this state.

Once it found condition time[dataout==1] to be true in motor state, control flow transfer to hotair state in which dataout value is initialized to 10, again same procedure.

Once during hotair state, if it is found the condition time[dataout==1] is to be true, then control transfers to the stopped state and this state treated as stop machine. Save the Simulink/Stateow model as .mdl file extension.

Convert to XML

After developing the SL/SF model in Matlab. It generates .mdl file. Next step is to convert that .mdl file into XML file using a command in Matlab. Figure 4.4 Shows the converted xml file of washing machine case study.

Generation of Dependency Graph

Next step is to apply our proposed algorithm of dependency graph generation and using an XML parser, convert the model into intermediate graph generation.

This algorithm takes .mdl file path as an input. After creating model objects for the SL / SF model and passes .mdl file path to the function graph, Generation and

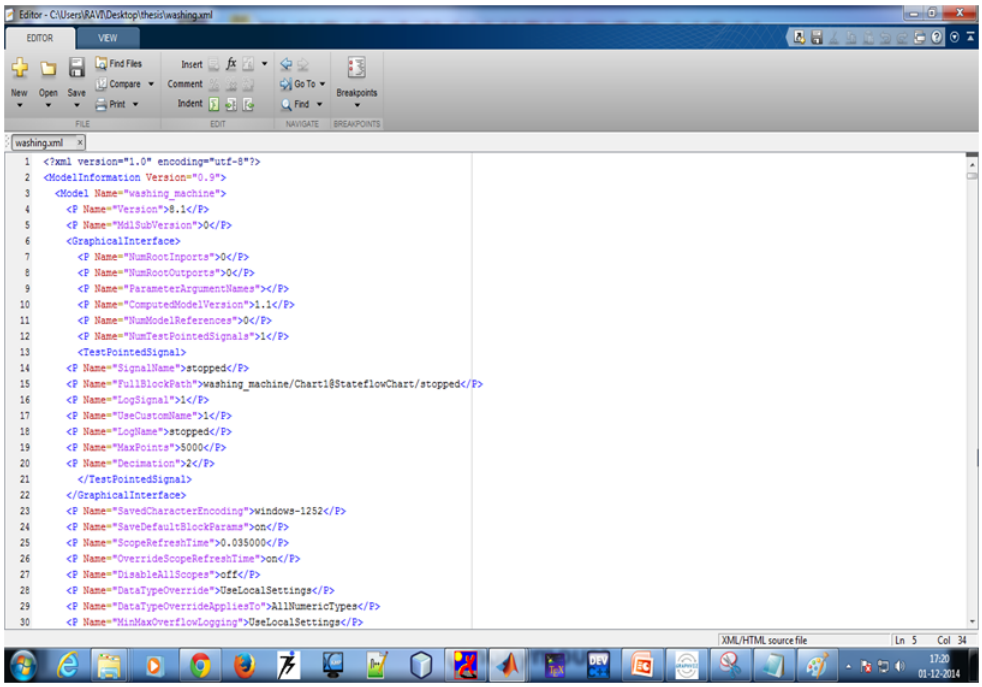

Figure 4.4: XML File of Automatic washing machine

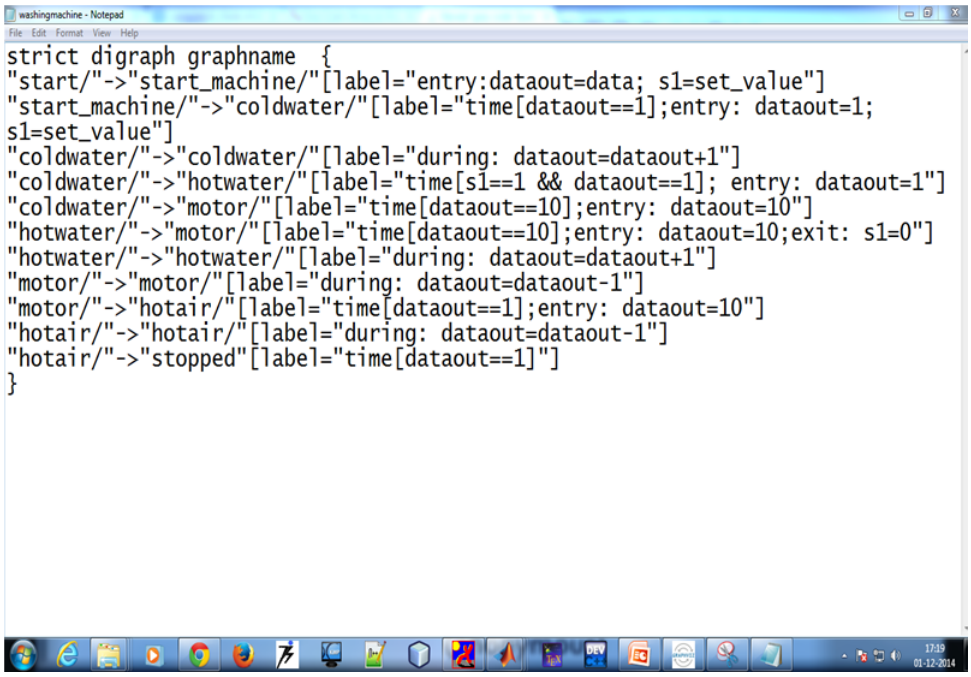

Figure 4.5: Dotty File of Automatic washing machine

Figure 4.6: Dependency Graph of Automatic washing machine

apply xml parser which takes an xml file as an input to xml parser. Next we have to apply loop up to all block covers in the model and within this loop, we have to perform these operations: read each block and extract the information of each block, then obtain all the next neighbor block of the current block, then Write the next block in the matrix form and store that into dot file. If any present block contains SL/SF subsystem, then push that block into the queue. After completing of first loop, we have to check whether the queue is empty or not, if queue is not empty, then again apply one loop up to block available in the queue or queue becomes empty and perform the following actions: read each block and extract the information of each block, then obtain all the next neighbor block of present block, then write the next block in the matrix form and store that into dot file. Figure 4.5 represents the dotty file that store blocks dependency which is further used to generate the dependency graph. GraphViz tool is used to generate dependency graph using generated dotty file. Figure 4.6 represents the SL/SF dependency graph for a model

washing machine.

Test Sequence and test case Generation

Next generating the graph, we have performed the DFS approach to generate the test sequences. These test sequences are used to generate test cases. In order to test the model efficiently, we have to create the test cases for each and every test sequence.

State Coverage: The coverage, which covers states in all the possible ways in the dependency graph is state coverage.

Transition Coverage: The coverage, which cover transition atleast once for all transition in the dependency graph, is called transition coverage.

The test Sequences for following case study of washing machine are :

Test Sequence 1:

Start→stopped(if signal is not generated).

Test Sequence 2:

Start → start-machine→coldWater→motor→hotAir→stopped

Test Sequence 3:

Start→start-machine→coldWater→hotWater→motor→hotAir→stopped

Next, for each test sequence we have to generate the test cases using the test sequences and dependency graph. Table 4.1 represents the set of test cases for SL/SF model.

Table 4.1: Test Cases of Automatic Washing Machine

Test ID	Current State	Input	Condition	Expected State
1	Start	dataout=data, s1=set_value		Start_machine
2	Start_machine	dataout=data, s1=set_value	Time[dataout==1] true	coldWater
3	coldWater	dataout = dadaout + 1	Time[dataout==10] false	coldWater
4	coldWater	dataout = 1	Time[dataout==1] true , Time[s1==1] true	hotWater
5	coldWater	dataout = 10	Time[dataout==10] true	motor
6	hotWater	dataout = dadaout + 1	Time[dataout==10] false	hotWater
7	hotWater	dataout = 10	Time[dataout==10] true	motor
8	motor	dataout = dadaout + 1	Time[dataout==1] false	motor
9	motor	dataout = 10	Time[dataout==1] true	hotAir
10	hotAir	dataout = dadaout - 1	Time[dataout==1] false	hotAir
11	hotAir		Time[dataout==10] true	stopped

Table 4.2: Information flow value for each state

State	Model_in Value	Model_out value	IF Vlue
coldWater	3	2	6
motor	2	3	6
hotWater	2	2	4
hotAir	2	2	4
Start_machine	1	1	1
Start	1	0	0
Stopped	0	1	0

Prioritization of Test cases

The prioritization of the generated test cases based on information flow (IF) value.

Steps of priortization of test cases:

Step 1: Compute model_in value for each node in the dependency graph.

Step 2: Compute model_out value for each node in the dependency graph.

Step 3: Calculate Information flow(IF) value for each node in the dependency graph.

Step 4: Test cases are prioritized based on the higher IF the value of the transition source state.

To compute for Model_in for each state we have to compare each state with all the transition destination nodes if the state matches than we increases the count and this counting continues till one iteration of transition destination nodes completed

and we store the count value of a state. This process continues till we compute for all states.

To compute Model_out for each state we have to compare each state with all the transition source nodes if the state matches than we increases the count and this counting continues till one iteration of transition source nodes completed and we store the count value in Model_out of a state. This process continues till we compute Model_out of all states.

For finding the IF value of each state we are computing the product of Model_in and Model_out of each state.

IF(A) = Model_in(A) x Model_out(A)
Where, Model_in(A)- Number of states calling state A. Model_out(A) - Number of states called by state A. IF (A) - Information flow value of state A.

State with higher IF value represents that the state having higher complexity, so the test cases are prioritized based on the higher IF the value of the transition source state.Table 4.2 represents the information flow value for the example of automatic washing machine. Table 4.3 shows the prioritized test cases after apply our approach.

Table 4.3: Prioritized Test Cases of Automatic Washing Machine

1	coldWater	dataout = dadaout + 1	Time[dataout==10] false	coldWater
2	coldWater	dataout = 1	Time[dataout==1] true , Time[s1==1] true	hotWater
3	coldWater	dataout = 10	Time[dataout==10] true	motor
4	motor	dataout = dadaout + 1	Time[dataout==1] false	motor
5	motor	dataout = 10	Time[dataout==1] true	hotAir
6	hotWater	dataout = dadaout + 1	Time[dataout==10] false	hotWater
7	hotWater	dataout = 10	Time[dataout==10] true	motor
8	hotAir	dataout = dadaout - 1	Time[dataout==1] false	hotAir
9	hotAir		Time[dataout==10] true	stopped
10	Start_machine	dataout=data, s1=set_value	Time[dataout==1] true	coldWater
11	Start	dataout=data, s1=set_value		Start_machine

Chapter 5

Computing Dependency using Slicing Approach

In this section, first we discuss about some basic terminology used in our works, definitions of some basic terms used in our work, proposed work of computing dependency using slicing approach, algorithms used in our approach, implementation with sample examples and their results.

5.1 Introduction

5.1.1 Simulink/Stateflow model

Matlab, Simulink is an extensively used notation in the development of dynamic system, especially for embedded system industry that allows models to be formed and exercised. Matlab, Simulink models are very often considered by industry as the architectural level design of software systems. The simulation amenities permit such models to be executed and observed. Since a real time control model can consist of a large number of blocks and states SL/SF model, so the complexity is more for these models and also debugging is difficult. So by computing the intra dependencies amongst the blocks in SL/SF using slicing approach can make the debugging easier.

5.1.2 Dependencies in SL/SF model

Simulink/Stateflow is an information stream situated graphical documentation where dataflow is by the structure and control stream must be ascertained.

We plan to discover how to focus information and control conditions for Simulink/Stateflow models.

Data dependencies in Simulink

Data stream in Simulink is given by the signal lines [14]. So data dependence can without much of a stretch be gotten from a watching the signal lines.
A block b2 is data dependent subject to block b1 if
i) b1 and b2 are associated by a signal line L, and
ii) L begins with a yield of b1 and finishes in an inputs of b2.

Control flow in Simulink

Fundamentally the control flow in Simulink [14] is displayed utilizing using subsystem:
i) Conditional subsystems: restrictive subsystems are enabled, activated, triggered, action and function-call subsystem.
ii) Loop subsystems: loops like while and for in Simulink are acknowledged by atomic subsystems.
iii) Multiport Switch and switch blocks: switch or multiport blocks can likewise be utilized to model the control flow in Simulink.

Control dependence in Simulink

In a model m containing the block b1 and b2, block b2 is control depenent on block b1 if
i)b1 is inside of a conditional execution context.
ii) b1 is the predicate block controlling the execution of b2.

Data dependence in Stateflow

Data dependence edges are utilized to speak to the dependence of a state of the data flow connected to a statflow model. This incorporates all the data variables utilized as a part of the entry, during, and exit section of any state.

The data variable can likewise take up with conditions, condition activities or transition action in any state transition.

Control Dependence in Stateflow

Control dependence in Stateflow relies on upon the moves between the states i.e transition between the states. The primary Stateflow execution begins from the default transition.

A control dependence edge speaks to on interstage dependence because of the change of the state in the Stateflow model. A control dependence may emerge in the accompanying three ways:

Case 1) Control dependence emerges when there is no transition mark connected with the transition between two states.

Case 2) When an active move from a state has a transition mark, then control dependence emerges between the states and the predicate block.

Case 3) Control dependence emerges when a state has way "exit action" and its outgoing transition is connected with condition action and transition action.

This is on account of once the state of a transition is fulfilled, then the condition transition makes place. Before executing the transition action and bouncing to the following next state, the way exit action of the current state happens after the transition execution.

This outcomes in control dependence emerging between the predicate and the state node with name having the name of the predicate node and state.

5.1.3 Slicing

Program slicing is a methodology for extracting the statements/blocks explanations of a program that influence or influences of the ideas of an arrangement of variables/blocks on particular details of interest in the course of study. Slicing is a useful method for decomposition and analysis of a model/system.

Slicing criterion

The point in the program and the variable/blocks of interest of point are usually brought up to as slicing criterion.

Simulink/Stateflow Slicing Criterion

A slicing criterion for Simulink/Stateflow model may be any block within the model with the exception of the subsystem or any state in the Stateflow model or any variable inside of the Stateflow.

Slice

The extracted statement/block in the program is called slice.

Simulink/Stateflow slice

A slice of a Simulink/Stateflow model m regarding slicing criterion c is a model m1 that
i) Contains just those blocks and states from the Stateflow that are significant to the slicing criterion c (forward slice and backward slice).
ii) Contains just those blocks and states from Stateflow to which the slicing model c is important and that care for the hierarchical structure of the Simulink/Stateflow model.

Forward slicing and Backward slicing with respect to Simulink model

Forward slicing: It depends on the direction of edges traversal (Forward edge traversal). Forward slices contain those blocks that are influenced or acted upon by the slicing criterion in the further execution of the model.

Backward Slicing: It depends on the direction of edge traversal (Backward edge traversal). Backward slices extract the blocks/states that influences the model at the point given by the slicing criterion.

5.2 Proposed Methodology for computing dependency using slicing approach

This approach uses dependency graph for representing forward and backward slicing where node presents the blocks of simulink model and edge represents the transition between the blocks of simulink model.

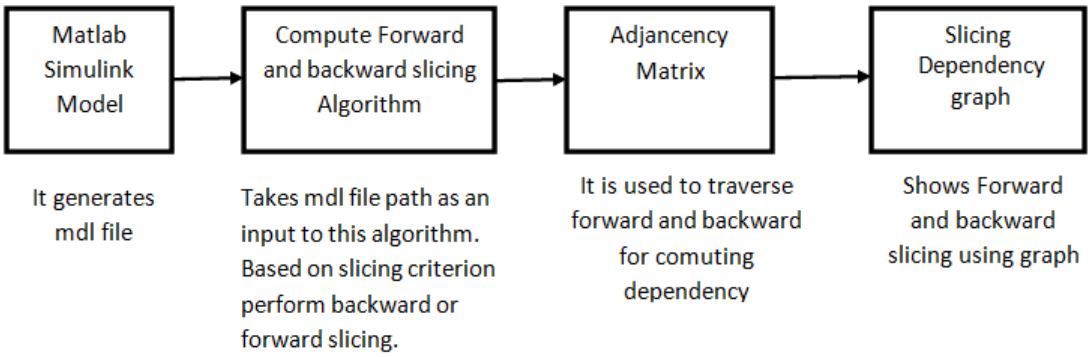

Figure 5.1: Block diagram of our proposed model for computing slicing shown using dependency graph

In this approach,shown in Figure 5.1, first develop the simulink model in Matlab that generates the .mdl file. Next we have to perform forward and backward slicing,

for performing this we have to pass mdl file path to our proposed algorithm of forward Slicing and backwardSlicing. Forward/backward slicing can be performed at a particular point of interest in the model called slicing criteria. Based on the slicing criteria we have to calculate the forward and backward slices. Forward slices contain those blocks that are influenced by the slicing criterion in the further execution of the model. Backward slices extract those blocks that influences the model at the point given by the slicing criterion. Four perform above slices we need an adjacency matrix of size NxN, where N represents the number of blocks in the Simulink models. The adjacency matrix contains the inter-dependencies amongst blocks of the model. Adjacency matrix is used to traverse forward and backward for computing dependency. With the help of matrix generated by our proposed algorithm, we have to generate a graph called dependency graph for both forward and backward slices shown by different colors.

5.2.1 Overall steps of our methodology

1) First, we develop a Simulink model using Matlab Simulink library. It generates a mdl file (model description language).
2) Read the blocks of Simulink model and assign a unique block id to each block.
3) Define slicing criterion at a particular point of interest.
4) Apply, our proposed algorithm of computing Forward Slices and Backward Slices which takes a .mdl file path as an input.
5) Compute interdependencies between the blocks and store in adjacency matrix.
6) Using adjacency matrix compute forward and backward slices.
7) Generate Dependency graph by converting matrix into graph form using the Jung library in Java.
8) Change the colors of computing slices block in dependency graph and finally generate the forward dependency slicing graph and backward dependency slicing graph.

5.2.2 Algorithm for Forward and Backward Slicing

In this section we discuss the detailed algorithm for calculating dependency between blocks of simulink models using forward and backward slicing approach.

Algorithm 3 ForwardSlicing

1: start main
2: Read the blocks of Sl/SF model.
3: Assign a block id to each block in the model.
4: Extract the list of edges(transition) between all the blocks
5: Create an adjacency matrix (`adj_mat`) of size M x M, where M is the number of blocks in SL/SF model and initialize `adj_mat` to 0.
6: Call function AdjanMatrix.
7: Get user input on which block forward slicing is performed(give a slicing criterion) and store it on IPblock
8: Pass adj_mat to jung API library so as to display the graph graphically.
9: Call function getpath (IPblock, `adj_mat`) // it returns Arraylist of nodes or blocks in the forward path
10: Store all the key values into a hashmap of returning Arraylist of the node from Getpath function.
11: Change the colors of all the nodes (keyvalues) present in the hashmap.
12: End main

Algorithm 4 Function:AdjanMatrix (block list, edge list, adj_mat)

```
1: start main
2: for all blobk 'i' do
3:     Get all destination blocks traversable directly from block i.
4:     for all  destination block 'j' do
5:         Adj_mat[i][j]=1;
6:     end for
7: end for
8: End main
```

Algorithm 5 Function: GetPath(IPblock, adj_mat)

```
1: start main
2: for all blobk 'i' do
3:     Using BFS concept to the directed dependency graph starting from IP block.
4:     List all nodes in the nodes or blocks present in path into Arraylist.
5:     Return Arraylist.
6: end for
7: End main
```

Algorithm 6 BackwardSlicing

1: start main
2: Read the blocks of Sl/SF model.
3: Assign a block id to each block in the model.
4: Extract the list of edges(transition) between all the blocks
5: Create an adjacency matrix (`adj_mat`) of size M x M, where M is the number of blocks in SL/SF model and initialize `adj_mat` to 0.
6: Call function AdjanMatrix.
7: Get user input on which block forward slicing is performed(give a slicing criterion) and store it on IPblock
8: Pass adj_mat to jung API library so as to display the graph graphically.
9: Calculate all the source node present in the graph.
10: Calculate all the path from different source nodes to the user_input node. Call findPath function.
11: Store all the different node present in the multiple path into the Arraylist.
12: Store all the Arraylist value (nodes) into the hashmap keyvalue.
13: Change the color of all the node which has key value of the hashmap.
14: End main

Algorithm 7 Function:AdjanMatrix (block list, edge list, adj_mat)

1: start main
2: **for all** blobk 'i' **do**
3: Get all destination blocks traversable directly from block i.
4: **for all** destination block 'j' **do**
5: Adj_mat[i][j]=1;
6: **end for**
7: **end for**
8: End main

Algorithm 8 Function: FindBackwardPath (Source node list, user_input node, adj_mat)

```
1: start main
2: for all adjacent node i of source node do
3:     if Check node i is not visited  then
4:         Set visited[node i] to true.
5:         Add node to path info stored into Arraylist path.
6:         call findPath(node i, user_input node, adj_mat) //recurcive call
7:     else
8:         if user_input node = =node i then
9:             Print path info stored in Arraylist path
10:            Clear path info
11:        end if
12:    end if
13: end for
14: End main
```

5.3 IMPLEMENTATION AND RESULT

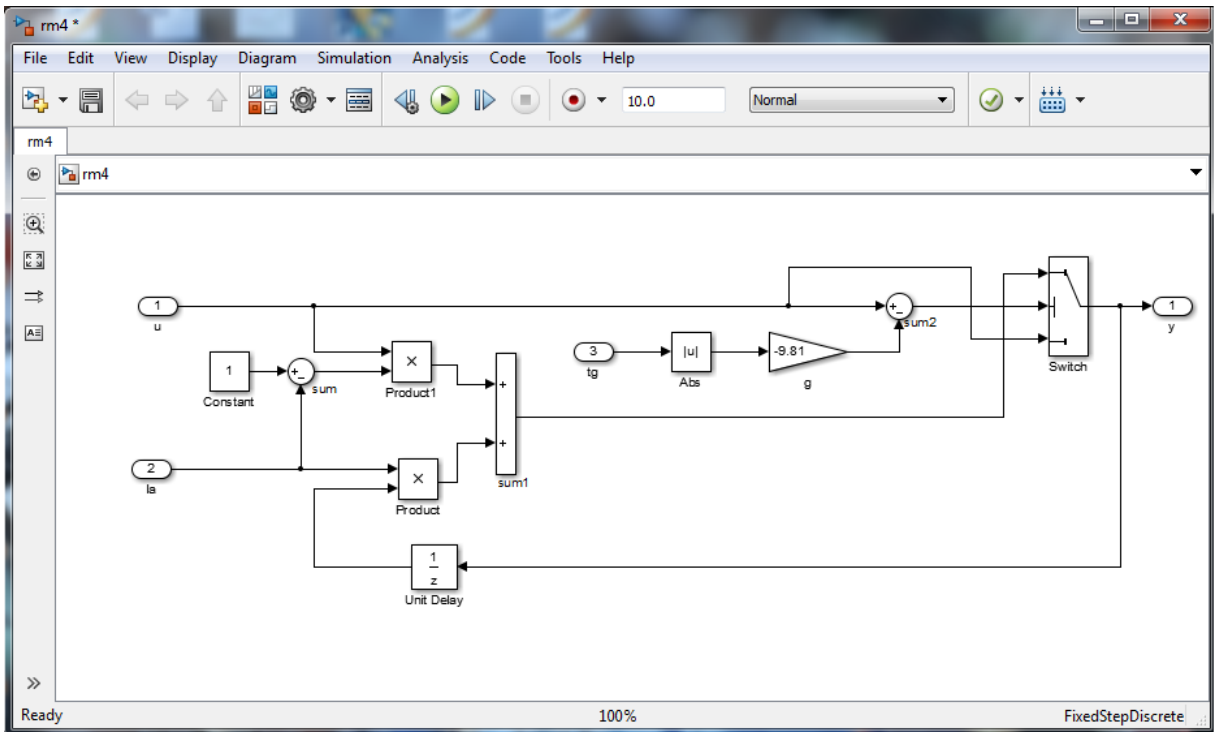

Figure 5.2: Sample Simulink Model

Apply our proposed algorithm for finding forward slicing and backward slicing dependency graph. Firstly, after applying mdl file path of model, the next step is to read all the blocks and assign a unique block id to each block present in the Simulink model.

The number of blocks along with a unique block for the Figure 5.2 of the sample Simulink model is as follows:

Number of blocks along with block id:

1 .rm4/g [Gain, 1:1]

2 .rm4/Switch [Switch, 3:1]

3 .rm4/Product [Product, 2:1]

4 .rm4/Abs [Abs, 1:1]

5 .rm4/Unit Delay [UnitDelay, 1:1]

6 .rm4/tg [Inport, 0:1]

7 .rm4/Product1 [Product, 2:1]

8 .rm4/Constant [Constant, 0:1]

9 .rm4/Sum [Sum, 2:1]

10 .rm4/u [Inport, 0:1]

11 .rm4/la [Inport, 0:1]

12 .rm4/Sum1 [Sum, 2:1]

13 .rm4/Sum2 [Sum, 2:1]

14 .rm4/y [Outport, 1:0]

Next step is to generate the dependency graph which captures the implicit dependency between the blocks in Simulink models.

The size of adjacency matrix is an N x N, where N is the total number of block present in a Simulink model. The adjacency matrix for the model in Figure 5.2 is as follows:

Adjacency matrix

00000000000010

00001000000001

00000000000100

10000000000000

00100000000000

00010000000000

00000000000100

00000000100000

00000010000000

01000010000010

00100000100000

01000000000000

01000000000000

00000000000000

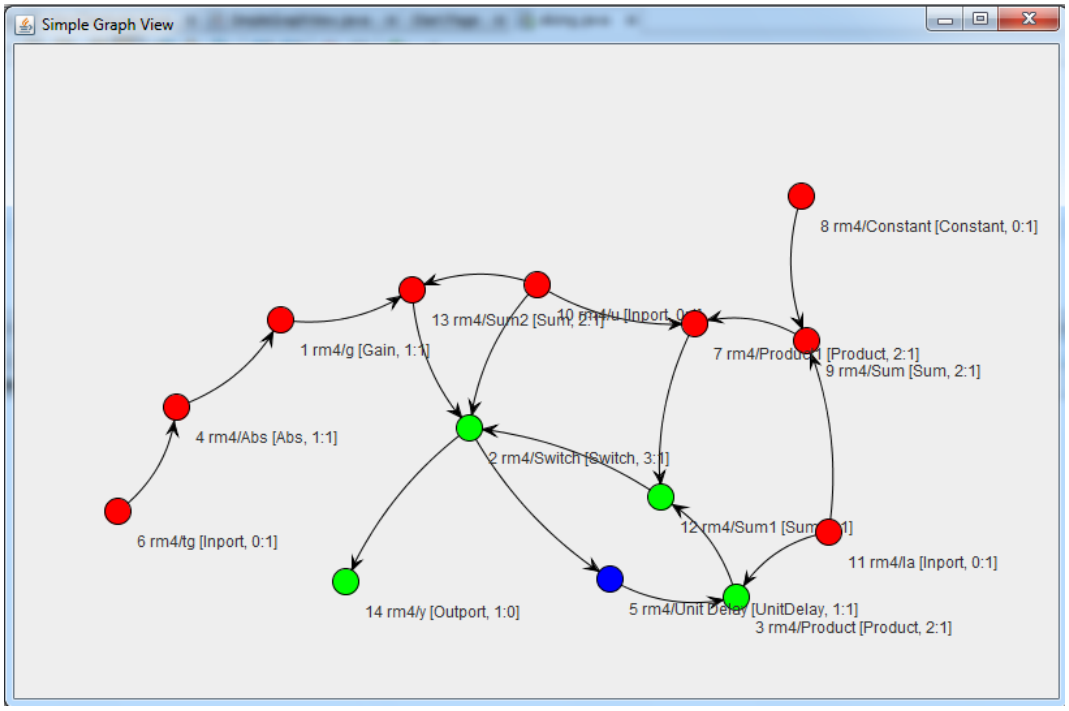

Figure 5.3: Dependency Graph showing forward slices

Next step is to ask for the user to define slicing criterion on which block we have to perform slicing:

Enter the block number where you want to find slice:

5

Forward slices

5.rm4/Unit Delay [UnitDelay, 1:1]

3. rm4/Product [Product, 2:1]

12. rm4/Sum1 [Sum, 2:1]

2. rm4/Switch [Switch, 3:1]

14. rm4/y [Outport, 1:0]

Next step is to generate the forward dependency graph using the junk library for representing graph and using an adjacency matrix. The forward dependency graph

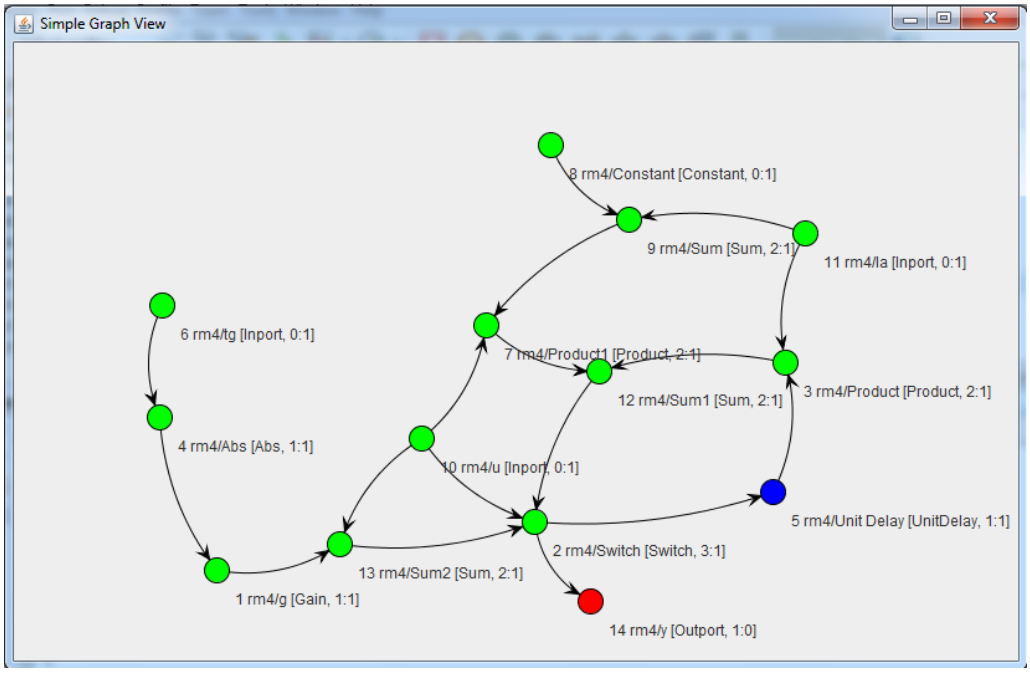

Figure 5.4: Dependency Graph Showing the backward slices

of the sample Simulink model is shown in Figure 5.3.

The next step is to find backward slicing. For finding backward slicing we need all the backward path from slicing criterion block. For this, firstly we need to calculate all the source node available in model or graph. Next step is to generate all the path from different source node to slicing criterion block using an adjacency matrix.

The different source node in the model sample Simulink model in Figure 5.2 are: Source 6, Source 8, Source 10, Source 11.

The different path from different source node to slicing criterion node is as follows:

Source 6

4→ 1→13→2→5

Source 8

9→7→12→ 2→5

Source 10

2→5

7→12→2→5

13→2 →5

Source 11

3→12→2→5

9→7→12→2→5

Next step is to collect all the node uniquely which is present in these different paths from source node to slicing criterion node and store it on hashmap key values are as:

Hashmap key value(Backward slices)

1 .rm4/g [Gain, 1:1]

2 .rm4/Switch [Switch, 3:1]

3 .rm4/Product [Product, 2:1]

4 .rm4/Abs [Abs, 1:1]

5 .rm4/Unit Delay [UnitDelay, 1:1]

6 .rm4/tg [Inport, 0:1]

7 .rm4/Product1 [Product, 2:1]

8 .rm4/Constant [Constant, 0:1]

9 .rm4/Sum [Sum, 2:1]

10 .rm4/u [Inport, 0:1]

11 .rm4/la [Inport, 0:1]

12 .rm4/Sum1 [Sum, 2:1]

13 .rm4/Sum2 [Sum, 2:1]

Next step converts the color of the key value node present in the hashmap that will represent the backward slicing in which colored node shows the affected node in backward w.r. t slicing criterion node.

Figure 5.4 represents the backward slicing dependency graph.

The main advantages of our approaches are: 1) A dependency analysis between the blocks of Simulink models for control and data dependence.

For dependence analysis, we demonstrate that a Simulink model control dependence can be derived from conditional execution contexts (CEC).

CECs are an implicit demonstration of the model which
1) Cannot be accessed (neither from the model file nor by MATLAB commands) and it Can be propagated to other blocks and different levels through the hierarchy.
2) It helps in realizing the model, helps in debugging the models and also assists in testing and sustenance of the model/system.

The computed slices must be precise and accurate to convey out the above actions.

The generated slices will be helpful in many applications like testing, software debugging, re-factoring of software, understanding, analysis of software ystem, maintenance, etc.

Chapter 6

CONCLUSION AND FUTURE WORK

MATLAB Simulink-Stateflow is a tool which is used for modeling dynamic systems, real time embedded system, Since simulink model may have several levels of hierarchy that can be viewed as different label of abstraction with several types of implicit dependencies between elements/blocks of the model that makes the model system more complex and difficult to perform any analysis on it and hence difficult to find bug and defects, so generation of test cases a bit difficult and complex. So we proposed a methodology to generate test cases and test sequences from Simulink-Stateflow models. Firstly we have developed the model in the MATLAB Simulink library environment by using Simulink/Stateflow designing tool. By simulation we verify the model. After verification by using our approach we generated a dependency graph. From that dependency graph we performed the DFS traversal operations and generated the test sequences. The test sequences are used to generate test cases.

Our approach covers much important coverage like state coverage and transition coverage. This is more accurate than the methods which are generating test cases using code generation.

We have also prioritized the generated test cases based on information flow value. It gives better understanding the test cases with higher involvement of the states within the model.

Our approach of a computing dependency graph using the static slicing approach is machinery on block levels and based on the dependency graph which makes easier to understand and analyze the model. With slicing, the complexity of a model can be compact to a specified point of concern by removing unrelated model elements. Using our approach of a computing the intra dependencies amongst the blocks in SL/SF using slicing approach can make the debugging easier.. The prioritization complexity of SL/SF model does not rely upon the number of discreate states, but depends on the number of simulation steps and number of continous variables.

We have generated test sequences for SL/SF model, moreover we plan to generate test cases for every embedded real time control system. We will also plan to optimize the test cases using genetic algorithm.In future visualization of Simulink/Stateflow model is also possible. In future it will be better to be proposed dynamic slicing approach on SL/SF model which will reduce the complexity of implementation of real time embedded system, also it will helps in analysis of design and implementation of real time embedded system.

Dissemination

1. **Ravikant Sharma** and Durga Prasad Mohapatra, *Prioritize Test Cases Generation for Simulink/Stateflow Models using Dependency Graph*, International Conference on Research Trends and Research Issues in Computer Science and Engineering (ICRTRICSE-2015),Sponsored by CSI and IEI, Andhra University College of Engineering, Visakahapatnam, Andhra Pradesh, India, May 2-3, 2015.

Bibliography

[1] The MathWorks, Mathworks MATLAB Simulink. "http://www.mathworks.com/products/simulink".

[2] T-vec,"Website. http://www.t-vec.com/ "

[3] Y. Zhan and Clark, "A search-based framework for automatic test-set generation for Matlabsimulink models," Software Eng. SE-10, PhD thesis, December 2005. University of York.

[4] Suraj Nayak. "A Metamodel for Simulink/Stateflow models and its applications", M. Tech. Thesis, IIT Kharagpur, Computer Science Department (2013).

[5] I. Sturmer and M. Conrad, "Test suite design for code generation tools," in Automated Software Engineering, 2003. Proceedings 18th IEEE International Conference on, pp. 286-290, IEEE, 2003.

[6] N. Vamshi Vijay, "Regression test selection based on analysis of Simulink/Stateflow models", M.Tech. Thesis, IIT Kharagpur, Computer Science Department (2012).

[7] A. Windisch, "A Search based testing of Simulink models containing stateflow diagrams,"IEEE Trans., vol. Software Engineering Companion Volume, pp. 395-398, 2009. Daimler Center for Automotive IT Innovations DCAITI, Tech. Univ. Berlin, Berlin, Germany.

[8] R. Systems, "Reactis simulator / tester." Website. http:www.reactive-systems.com.

[9] A. A. Gadkari, S. Mohalik, K. Shashidhar, A. Yeolekar, J. Suresh, and S. Ramesh, "Automatic generation of test-cases using model checking for SL/SF models," in 4thMoDeVVa workshop Model Driven Engineering, Verification and Validation, p.33, 2007.

[10] Bates, Samuel, and Horwitz, Susan. "Incremental program testing using program dependence graphs", In Proceedings of the 20th ACM SIGPLAN SIGACT symposium on Principles of programming languages, (1993), pp. 384-396.

[11] Adepu Sridhar, "Generating Test Sequences and Slices for Simulink/Stateflow Models",M. Tech thesis,Computer science and engineering department, NIT Rourkela, 2013.

[12] M. Li and R. Kumar, "Model-based automatic test generation for simulink/stateflow using extended finite automaton," in Automation Science and Engineering (CASE),2012 IEEE International Conference on, pp. 857-862, IEEE, 2012.

[13] Ray, Rajarshi, "Automated translation of matlab Simulink/Stateflow models to an intermediate format in hyvisual,"MSC degree, thesis, Chennai Mathematical Institute, Computer Science Department (2007).

[14] Reicherdt, Robert, and Sabine Glesner. "Slicing MATLAB simulink models", Software Engineering (ICSE), (2012) 34th International Conference on, pp. 551-561. IEEE, (2012).

[15] Aditya Agrawal, Gyula Simon, Gabor Karsai," Semantic Translation of Simulink/Stateow Models to Hybrid Automata Using Graph Transformations", Electronic Notes in Theoretical Computer Science 109: 43-56 (2004)

[16] G. N. Binkley, David and M. Harman, " An empirical study of static program slice size," ACM Transactions on Software Engineering and Methodology, vol. 16, Apr. 2007.

[17] B. Korel, I. Singh, L. Tahat, and B. Vaysburg, "Slicing of state-based models," in Software Maintenance, 2003. ICSM 2003,Proceedings. International Conference on,pp. 34-43,IEEE, 2003.

[18] S. Van Langenhove and A. Hoogewijs,"System verification through logic tool support for verifying sliced hierarchical statecharts," pp. 142-155, 2007,Springer Berlin Heidelberg.

[19] D. W. Ji, Wang and Q. Zhi-Chang,"Slicing hierarchical automata for model checking Uml Statecharts," (London, UK), pp. 435-446, in Proceedings of the 4th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering, 2002.

[20] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," pp. 177-184, in Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 1984.

[21] T. R. Horwitz, Susan and D. Binkley, "Interprocedural slicing using dependence graphs," ACM Transactions on Programming Languages and Systems, vol. 12, pp. 26-60, 1990.

[22] Dalal, Siddhartha R., Ashish Jain, Nachimuthu Karunanithi, J. M. Leaton,Christopher M. Lott, Gardner C. Patton, and Horowitz, Bruce M. "Model based testing in practice." In Proceedings of the 21st international conference on Software engineering, pp. 285-294. ACM, (1999).

[23] Bringmann, Eckard, and Kramer Andreas, "Model-based testing of automo tive systems," In Software Testing, Verification, and Validation, 2008 1st International Conference on, pp. 485-493. IEEE, (2008).